PPLib: toward the automated generation of crowd computing programs using process recombination and auto-experimentation

De Boer, Patrick M ; Bernstein, Abraham

Abstract: Crowdsourcing is increasingly being adopted to solve simple tasks such as image labeling and object tagging, as well as more complex tasks, where crowd workers collaborate in processes with interdependent steps. For the whole range of complexity, research has yielded numerous patterns for coordinating crowd workers in order to optimize crowd accuracy, efficiency, and cost. Process designers, however, often don't know which pattern to apply to a problem at hand when designing new applications for crowdsourcing. In this article, we propose to solve this problem by systematically exploring the design space of complex crowdsourced tasks via automated recombination and auto-experimentation for an issue at hand. Specifically, we propose an approach to finding the optimal process for a given problem by defining the deep structure of the problem in terms of its abstract operators, generating all possible alternatives via the (re)combination of the abstract deep structure with concrete implementations from a Process Repository, and then establishing the best alternative via auto-experimentation. To evaluate our approach, we implemented PPLib (pronounced "People Lib"), a program library that allows for the automated recombination of known processes stored in an easily extensible Process Repository. We evaluated our work by generating and running a plethora of process candidates in two scenarios on Amazon's Mechanical Turk followed by a meta-evaluation, where we looked at the differences between the two evaluations. Our first scenario addressed the problem of text translation, where our automatic recombination produced multiple processes whose performance almost matched the benchmark established by an expert translation. In our second evaluation, we focused on text shortening; we automatically generated 41 crowd process candidates, among them variations of the well-established Find-Fix-Verify process. While Find-Fix-Verify performed well in this setting, our recombination engine produced five processes that repeatedly yielded better results. We close the article by comparing the two settings where the Recombinator was used, and empirically show that the individual processes performed differently in the two settings, which led us to contend that there is no unifying formula, hence emphasizing the necessity for recombination.

DOI: https://doi.org/10.1145/2897367

DOI: https://doi.org/10.1145/2897367

PPLib: Towards the Automated Generation of Crowd Computing Programs using Process Recombination and Auto-Experimentation[1]

PATRICK M. DE BOER, University of Zurich
ABRAHAM BERNSTEIN, University of Zurich

Crowdsourcing is increasingly adopted to solve simple tasks such as image labeling and object tagging as well as more complex tasks, where crowd workers collaborate in processes with interdependent steps. For the whole range of complexity, research has yielded numerous patterns for coordinating crowd workers in order to optimize crowd accuracy, efficiency, and cost. Process designers, however, often don't know which pattern to apply to a problem at hand when designing new applications for crowdsourcing.

In this paper, we propose to solve this problem by systematically exploring the design-space of complex crowd-sourced tasks via automated recombination and auto-experimentation for an issue at hand. Specifically, we propose an approach to finding the optimal process for a given problem by defining the deep structure of the problem in terms of it's abstract operators, generating all possible alternatives via the (re-)combination of the abstract deep structure with concrete implementations from a Process Repository, and then establishing the best alternative via auto-experimentation.

To evaluate our approach, we implemented PPLib (pronounced "People Lib"), a program library that allows for the automated recombination of known processes stored in an easily extensible Process Repository. We evaluated our work by generating and running a plethora of process-candidates in two scenarios on Amazon's Mechanical Turk followed by a meta-evaluation, where we looked at the differences between the two evaluations. Our first scenario addressed the problem of text translation, where our automatic recombination produced multiple processes whose performance almost matched the benchmark established by an expert translation. In our second evaluation, we focused on text shortening; we automatically generated 41 crowd process candidates, among them variations of the well-established Find-Fix-Verify process. While Find-Fix-Verify performed well in this setting, our recombination engine produced 5 processes that repeatedly yielded better results. We close the paper by comparing the two settings where the Recombinator was used, and empirically show that the individual processes performed differently in the two settings which lead us to contend that there is no unifying formula, hence emphasizing the necessity for recombination.

Categories and Subject Descriptors: **Human-computer interaction (HCI)**: Interaction techniques

General Terms: Process design, Process recombination, Crowd Sourcing, Collective Intelligence, Auto-Experimentation

Additional Key Words and Phrases: Human computation algorithms

ACM Reference Format:

Patrick M. de Boer, Abraham Bernstein, 2016. PPLib: Towards the Automated Generation of Crowd Computing Programs using Process Recombination and Auto-Experimentation. *ACM Trans. Intelligent Systems and Technology*, 4, Article 39 (March 2010), 6 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Accelerated by the industrial revolution, work was increasingly structured into processes, where a person or a group of people fulfill very specialized components of the overall production to be more efficient. In crowdsourcing, we can observe a comparable division of labor emerging, where we structure crowd workers into crowdsourcing processes that aim to get work done as accurately, efficiently and inexpensively as possible. To enable this, a number of best practices (or *crowdsourcing patterns*) have been established, including Find-Fix-Verify [Bernstein et al. 2010], majority voting, and iterative refinement [Little et al. 2010].

[1] An abstract about this work was presented at the 2015 Collective Intelligence Conference

However, there are currently no guidelines on when to use which pattern, or even on how many crowd workers should be employed at each step. Essentially, system designers are left to pick and implement one of the patterns proposed *purely on intuition* without any assurance that it is an optimal or even suitable choice.

In the realm of Business Process Reengineering (BPR), [Bernstein et al. 1999] introduced the *Process Recombinator* through which novel business processes could be generated from a repository of existing processes. As a first step, their approach calls for process designers to identify the process deep structure – i.e., the main abstract processing steps that make up the core activities of the process. They then rely on the paradigm of specialization from object-oriented programming and F-logic [Angele et al. 2009] to enrich these descriptions into concrete instantiations of the desired processes. This approach regards the deep structure of the process as a specification of the design space [Ulrich and Eppinger 1995] and the recombination into its instantiations as a systematic exploration of this design space. Unfortunately, given the Process Recombinator's application domain in BPR, it did not generate executable processes, as this would involve reorganizing corporations – a sizable effort that is hardly practical. Hence, the authors suggested evaluating the results by BPR-specialists.

In realm of crowdsourcing, it is much easier to try different processes structures for a problem at hand. Hence, we extend the idea of process recombination with a dependency injector [Fowler 2004] that generates executable crowd programs that can then be automatically tested and evaluated in a crowdsourcing market such as Amazon Mechanical Turk[2]. Specifically, we propose to not only systematically generate the whole design space of crowd programs as specified by a process deep structure, but also to automatically run experiments that determine which alternative is the best[3] choice – thereby "reducing" the problem of designing complex crowdsourcing solutions to that of defining their deep structure and then letting the recombination and auto-experimentation process decide, which instantiation promises to be the most successful.

As a consequence, the main contributions of this paper are as follows:

- We propose a novel methodology for systematically exploring the design space of crowd-computing processes that combines process-recombination with auto-experimentation.
- We introduce a system called PPLib (a recursive acronym that stands for PPLib Programming Library) that incorporates this methodology. By leveraging the PPLib Process Repository (PPR)—a systematically organized repository of crowd-process patterns and process fragments—PPLib is capable of generating all combinations of the different process implementations to exhaustively search and evaluate the space of candidate processes in order to **find the optimal[3] process for a given problem**.
- We introduce a first version of our repository of existing abstract crowd processes that can be used as building blocks for recombination.

[2] http://www.mturk.com
[3] The measure of optimality can be defined by the user through a utility function. Popular defaults for such a utility function may be an optimization on cost, duration, and adequacy of the result.

- We release PPLib and PPR under an Open Source License on GitHub[4], where we also provide documentation and show how applications can (re-)use our approach, our software and our process repository.

We evaluate our methodology and its associated library by showing that PPLib is capable of independently designing and finding non-trivial, well-performing, robust crowd processes for well-known crowd-computing problems with objective solutions (i.e., tasks with correct or incorrect outcomes) such as text translation and text shortening.

2. RELATED WORK

2.1 Business Process Reengineering

In the late 1990's the topic of business process reengineering (BPR) and management [Hammer and Champy 1993] rose to great prominence. BPR was seen as an approach to modernize organizational processes in corporations and turn them into more customer focused institutions. Extending on ideas of the Tayloristic method and early approaches to high-level process automation [Hammer et al. 1977; Zisman 1978] numerous business process modeling approaches were proposed [Lee et al. 2008]. These modeling approaches still left the question of how to systematically innovate new approaches. [Malone et al. 1999] proposed the use of a Process Handbook, an electronic catalogue of organizational processes, to inspire innovation. The MIT Process handbook [Thomas W. Malone et al. 2003] encompassed 5000 business process that were organized according to principles of coordination theory [Malone and Crowston 1994] as well as a specialization hierarchy, which used a special type of inheritance to organize processes in a taxonomy [Battle et al. 2005]. The advantage of this organization was that non-BPR-specialists were able to browse the handbook for inspiration and add novel processes.

Inspired by product design and management approaches [Ulrich and Eppinger 1995], Bernstein *et al.* [Bernstein et al. 1999] proposed to use the repository of the process handbook as basis to automatically recombine different elements of processes to be able to explore the whole design space of business processes. Specifically, they proposed that innovators define business processes using abstract processes from the process handbook, and then established that a Process Recombinator would be able to generate all processes of this design space. The resulting process candidates could then be used as an inspiration to establish innovative and new processes in companies - shifting the main focus of BPR from process design to choosing among the most useful alternatives. The authors concede that one of the main limitations of their work is that process recombination may lead to too many candidates and that it is difficult to support an automated pruning of the candidates in the realm of BPR.

2.2 Crowdsourcing

Crowdsourcing describes the process of asking crowds, i.e. people on the web, to answer questions and provide work – sometimes in exchange for money.

Academia has been provided many innovative methods to engage with such highly available on-demand workers. One of the influential thoughts for this publication particularly was to view a crowd worker as a human computer, which led [Little et al.

[4] https://github.com/uzh/PPLib

2010] to present Turkit, a scripting language that allows its programmers to easily interact with crowd workers on Amazon's Mechanical Turk platform. In particular, they focused on how the high latency of human computation affects writing and debugging such code, which led them to propose the Crash&Rerun programming pattern, allowing the programmer to repeatedly rerun and debug processes without needing to republish costly previous steps. [Franklin et al. 2011] used a similar approach in CrowdDB to delegate joins to human workers. [Tranquillini et al. 2015] developed CrowdComputer, a programming language for crowdsourcing and equipped it with a visual editor to easily create crowd processes in BPR. Some areas within crowdsourcing have seen the advent of tailored frameworks: Medusa [Ra et al. 2012] poses an example for the area of crowd sensing.

Another group of frameworks focus on the Map-Reduce approach to manage highly interdependent tasks. [Kittur et al. 2011]'s CrowdForge is among the main constituents of this group. They split down large problems into sub-problems, using either algorithms or other crowd workers. Once these sub-problems are solved by machine agents or human agents, the results get collected and aggregated.
[Ahmad et al. 2011] expanded on this idea and proposed Jabberwocky, which added the possibility of using a powerful high-level procedural language to process tasks.

Programming in a computer language to essentially coordinate human workers and CPUs equally led [Bernstein et al. 2012] to think of the Global Brain–metaphor, where they combine networked humans and computers in a heterogeneous graph. Automan [Barowy et al. 2012], an automatic crowd programming system, fits this metaphor well and integrates human computation tasks as function calls in a standard programming language, hereby blurring the lines between CPUs and human processors. Among the main problems with programming the global brain is the fundamental difference between humans and computers in terms of motivational, error, and cognitive diversity [Bernstein et al. 2012]. Furthermore, many problem-solving processes are difficult to specify ex-ante and only gain more specific definitions during executions. [Bernstein 2000] proposed the notion for tasks to move along *the specificity frontier*, from well-defined and static to loosely defined and dynamic. While Automan requires processes to be very well-defined, which therefore leads to static processes, applications developed using Turkomatic [Kulkarni et al. 2012] are located on the opposite end of the specificity frontier. In Turkomatic, a programmer specifies her high-level goal in plain English and crowd workers are then employed to formalize the description – and execute it.

In order to address the diversity of human error, research has identified quality assurance as an important problem in crowdsourcing. Automan, for example, treats answers of multiple choice queries as samples of a statistical process and continues sampling until a predefined confidence-value is reached. Similarly, Beat-By-K continues sampling until the item receiving the most votes has at least K more votes than the second placed item [Goschin 2014]. [Livshits and Mytkowicz 2014] calculate the right amount of samples to be taken for each query ex-ante using power analysis. [Bernstein et al. 2010] proposed the popular Find-Fix-Verify pattern, in which workers would first be presented with a list of options where they mark entries that are to be refined. In the Fix step, a number of workers generate alternatives to these entries, out of which the best one is chosen in the verify step. They have deepened the idea of Find Fix Verify in Context Trees [Verroios and Bernstein 2014], where work is divided & conquered on a tree-like structure. In Iterative Dual-Pathway [Liem and Chen 2011], workers are assigned to one of two groups. Both groups are working on the same tasks in parallel but don't see the other group's answers to the same task. Results are then verified by comparing the answers of the two groups for a single task. [Inel et al. 2014] proposed CrowdTruth, a framework that uses the disagreement between crowd workers to evaluate data quality, question ambiguity and worker quality.

We have proposed CrowdLang [Minder and Bernstein 2012a], a workflow-based language that enabled *manual* recombination processes. This contribution expands on our earlier idea by presenting an *automated* recombination mechanism that works on a process repository and combines it with auto-experimentation for selecting successful crowd programs.

2.3 Auto-Experimentation

Automating science is a long-standing dream of scientists in many disciplines. The closest to full automation of the scientific endeavor is probably the Robot Scientist called "Adam" [King et al. 2009], which autonomously generates hypothesis, plans and executes experiments to test the hypotheses, and draws conclusions to discover novel findings in yeast genetics. While in most other domains of science such a fully automated discovery process is somewhat futuristic, many domains have adopted automation approaches for a variety of steps in the scientific process. In particular the automation of experimentation is very prominent in disciplines ranging from the computational sciences, where experiments are completely virtual, to chemistry, where combinatorial testing allows running multiple experiments in parallel. In Knowledge Discovery in Databases (KDD) and statistical analyses, for example, people have explored the notion of Intelligent Discovery Assistants [Bernstein et al. 2005; Serban et al. 2013] – systems that advise human data analysts on which approaches to use in experimentation. Furthermore, this domain has a long tradition to explore alternatives automatically via auto-experimentation: the automatic planning, execution, and evaluation of experiments (e.g., [Serban 2010]).
In crowdsourcing, auto-experimentation is most often used to evaluate hypotheses of researchers (e.g., [Sheng et al. 2008]). Crowd programs are scheduled automatically, and evaluated either automatically (through the use of yet other crowd programs) or manually (by a panel of experts). We have, however, not found any approach that proposes to pair the systematic exploration of the design space of a solution with an evaluation of the candidates via auto-experimentation.

3. THE PPLIB METHODOLOGY

Composing crowdsourcing solutions is essentially a human problem-solving task. According to [Newell and Simon 1972; Simon 1977] human problem solving can be organized into four phases: (i) intelligence: defining a problem, (ii) design: composing a variety of alternative choices for the problem, (iii) choice: the evaluation of the alternative solutions to select the best alternative, and (iv) implementation: the implementation of the chosen solution. Inspired by [Bernstein et al. 1999], we modeled our PPLib methodology on Simon's phases.

1. *Intelligence*: *Identifying the process deep structure.* Simon's first phase of human problem solving calls for the identification and definition of a problem. In AI

planning, a generic problem definition is typically given via the specification of the initial and desired state as well as the domain model (i.e., the set of possible actions in the problem domain). In our case we propose to define a crowdsourcing problem via the specification of its deep structure [Chomsky 1965] – its most abstract activities akin to the top level of a grammar in Hierarchical Task Network (HTN) planning [Nau 1987]. Specifically, we propose to define the deep structure of the solution by incorporating building blocks chosen from the PPLib Process Repository (PPR) and/or from a programming language. Elements chosen for the ontology can later be used for recombination [Bernstein et al. 1999]. Note that the HTN-inspired approach somewhat blurs the boundary between the first two steps of Simon's approach, as identifying the deep structure of a solution specifies the *design space* for them, since abstract elements chosen from the PPR indicate the degrees of freedom that can be exploited for process recombination and, hence, for the design of solution alternatives.

2. *Design: Defining parameter candidates.* Our choice of problem specification via process deep structure already outlines the structure of the design space. PPR processes can be configured by supplying parameters such as crowd worker count, confidence value to be achieved in repeated single choice questions, or money to be paid to crowd workers for the tasks involved. Hence, the first element of *design* in our methodology encompasses the specification of parameter ranges. The second element requires the definition of a utility function to rank the solutions of the process candidates resulting from the auto-experimentation phase.
3. *Choice: Recombination and Auto-Experimentation.* In order to be able to choose between the possible solution candidates, they need to be instantiated and their performance needs to be evaluated. To fulfill this goal, we propose to explore the design space of possible solutions as follows:
   (i) Generate the recombinations. In this first step we take the problem definition specified in steps 1 & 2 and generate all possible instantiations employing the deep structure and the PPR like a production grammar to create surface structures.
   (ii) Run auto-experimentation. This second step takes all generated surface structures, runs them, and evaluates the results using the utility function defined in step 2. Note that in the simplest case, the ranking implied by the utility function would indicate one winning process, while an extended evaluation may find multiple solutions with different tradeoffs (such as robustness to various conditions, etc.).
4. *Implementation: Deployment of the chosen process*. Implement the chosen process in the production environment.

This abstract definition of our methodology leaves a number of questions open. For that reason the remainder of this section will further explain the main elements of the methodology, before we cover implementation details for each of these steps in Section 4, where we also show an example of how to use this abstract definition.

3.1 The PPLib Process repository

The PPLib Process Repository (PPR) stores the crowd computing, human interaction, and general process organization patterns and organizes them in a taxonomic structure. Inspired by the MIT Process Handbook [Malone et al. 1999], we chose to organize them into a type hierarchy, where more special processes can fulfill the role of a more generic ones [Wyner and Lee 2002]. In the style of object-oriented programing we call more general entries that are not sufficiently specified to be enacted (or executed) *abstract crowd processes* and enactable entries in the PPR *concrete crowd processes*.

The PPR's structure is inspired by both the Process Handbook [Thomas W Malone et al. 2003; Malone et al. 1999] and the collective intelligence genomes [Malone et al. 2010]. The PPR currently contains the top-level genomes suggested in the WHAT dimension of their ontology (see also Figure 1): CREATE and DECIDE. CREATE is used for different ways of getting crowd workers to create collections of items such as texts. DECIDE is used to select one (or more) element from a collection of alternatives.
All processes in PPR have a specified input type and a specified output type, which also get inherited to ensure polymorphism. Since the PPR builds upon inheritance and polymorphism, one can easily build complex processes that consist of other processes in the PPR.
Note that the PPR does not claim to be a complete library of all crowdsourcing tasks. The Process Handbook, e.g., on which the PPR is based also suggests COMBINE or DIVIDE as top level processes. We see it as a growing repository whose usefulness increases as people add more building blocks.

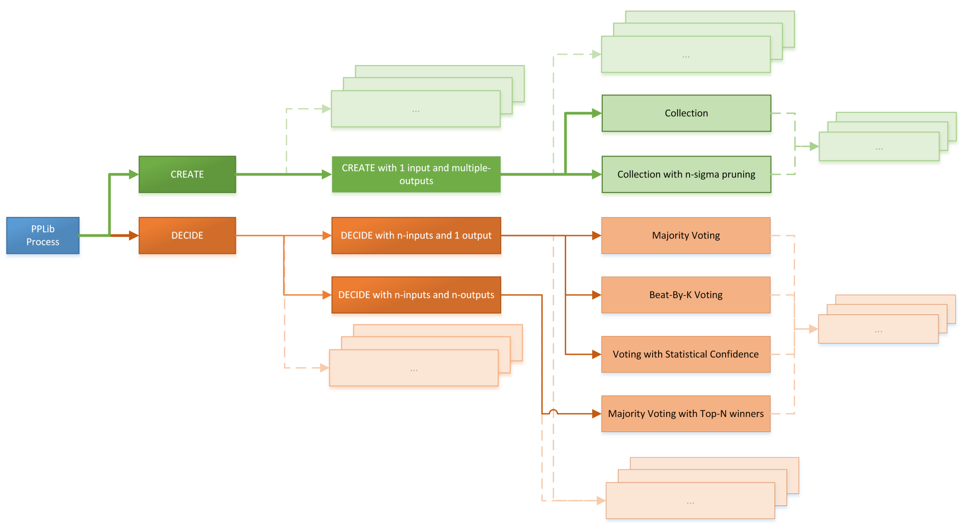

Figure 1: Top Level of the PPLib Process Repository including the basic building blocks

3.2 Process Recombination

Having established the notion of abstract processes in section 3.1, we would now like to show how these abstract processes are used for recombination. We distinguish between 2 forms of recombination: parameter recombination and workflow recombination.
All processes in the Process Repository are configurable using parameters. Often, one would like to try different variants of each individual process, such as different payment values, different questions to ask crowd workers, or varying numbers of worker counts used. For discrete parameter-types, the PPLib Recombinator

generates all possible combinations of the specified parameters of an abstract process, where each combination results in its own instance of the abstract process under recombination.

Workflow recombination leverages the plug-in nature of the specialization hierarchy of PPR to generate alternatives. Consider the case, when multiple Human Computation Tasks are declared in the application under recombination, e.g., $HComp_1$ and $HComp_2$. Here, the Workflow Recombinator looks for all possible alternatives that the PPR offers for both $HComp_1$ and $HComp_2$ and generates them. If, for example, the PPR specifies process P_1 and P_2 for $HComp_1$ and process P_3 and P_4 for $HComp_2$, then the instantiations $\{ <P_1, P_3>, <P_1, P_4>, <P_2, P_3>, <P_2, P_4> \}$ are generated. Note that this process is recursive. Hence, if P_2 contains the steps $P_{2.1}$ and $P_{2.2}$, where $P_{2.1}$ is an abstract process that has two alternatives $P5$ and $P6$ then the list above is extended to the following resulting processes: $\{ <P_1, P_3>, <P_1, P_4>, < < P_5 , P_{2.2} >, P_3>, < < P_6 , P_{2.2} >, P_3>, < < P_5 , P_{2.2} >, P_4>, < < P_6 , P_{2.2} >, P_4> \}$.

It is evident that generating all combinations of processes with their parameters has exponential runtime and space complexity – this requires a process designer to think carefully about which parameters and/or workflows to recombine. Hence, we also allow process designers to specify restrictions on which recombinations are actually used by specifying parameter ranges or valid specializations explicitly in the deep structure.

3.3 Auto-experimentation

Once all requested process candidates are generated, they can be evaluated in parallel by simply passing each candidate to the process deep structure that has been formalized (e.g., into program code) before and collect the results.

The auto-experimentation engine executes all recombined processes, retrieves their results, and applies the user-defined utility function to it in order to rank them. More formal, given a set of recombined crowd-processes P, a specified number of iterations the experiment should be repeated i, and a utility function f: Execute each process $p \in P$ i times and apply the utility function f to each result obtained in the individual iterations. In the end, rank all p by their mean[5] utility as evaluated by f across all i.
The sample size for each p is therefore determined by i and should be chosen large enough, such that one can derive scientifically sound findings from the results – which often leads to a tradeoff between the cost and accuracy of the experiment.
Please note that PPLib leaves the definition of f entirely to the user. Popular examples include throughput time, money spent, and some quality measure such as text length for a crowd task of text shortening.

4. THE PPLIB PROGRAMMING LIBRARY

We have implemented each of the aforementioned core modules into PPLib, an extensible programming library that supports computer-crowd worker interaction.

The PPLib Programming Library is modeled after the generic human communication patterns described in speech acts [Searle 1969; Winograd and Flores 1986]. Comparable to Turkit [Little et al. 2010], one develops code that asks crowd workers questions and acts upon their answers. More complex interactions can be aggregated

[5] Other aggregation functions suitable to optimization (such as median) could be used here as well.

into reusable Crowd Processes relying on the principle of stepwise refinement [Wirth 1971] to keep code clean and simple.
PPLib is comprised of several components that work together as follows (Figure 2):

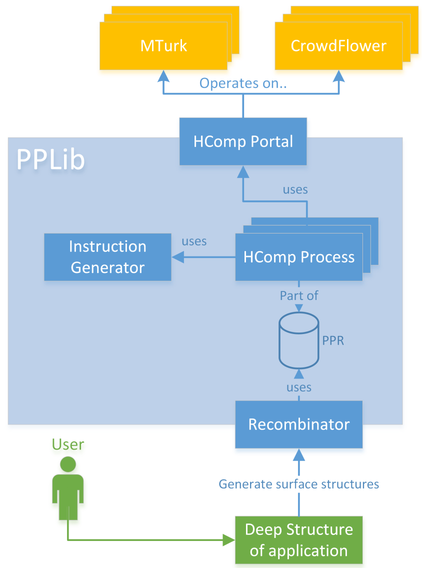

Figure 2: PPLib architecture

- **Recombinator:** Generates process surface structures based on the search space definition provided by the user's deep structure definition and constraints. While it is possible to use other components of PPLib directly, most users will only interact with the Recombinator.
- **HComp Process:** Central components within PPLib. They expose interfaces with lists of parameters that need to be supplied for the process to be instantiated.
- **PPR (PPLib Process Repository):** The collection of HComp Processes.
- **Instruction Generator:** Creates Instructions for crowd workers based on user defined recipes.
- **HComp Portal:** Abstracts communication with portals (e.g., MTurk) such that processes can be portal agnostic.

PPLib is written in Scala[6] and can therefore easily be integrated in any existing Java or Scala application using industry standard dependency managers such as SBT[7](Apache Ivy[8]) or Apache Maven[9].

4.1 Human Computation Portals in PPLib

PPLib is built to be Human Computation Platform independent, i.e. programs using PPLib as an intermediary will run on any supported Human Computation Platform. PPLib comes with built-in support for CrowdFlower[10] and MTurk. Switching a process from one portal to another is very easy for PPLib applications and can be done by changing a parameter value of crowd process. One can even use workers from different portals in the same process and effectively have them interact with each other, transcending the borders posed by such portals.
Furthermore, PPLib is built in a way that makes it easy to add support for other existing and new platforms. Hence, support for other paid- and unpaid platforms such as UpWork[11] or Facebook[12] could be added.

[6] http://www.scala-lang.org/
[7] http://www.scala-sbt.org/
[8] http://ant.apache.org/ivy/
[9] http://maven.apache.org/
[10] http://www.crowdflower.com
[11] http://www.upwork.com
[12] http://www.facebook.com

4.2 Human Computation Processes in PPLib

A Human Computation Process in PPLib is any (Scala/Java) class inserted into the PPR using our ontology (see section 3.1) and implementing the defined interface methods.
To support long running transactions, PPLib includes reusable building blocks of the *crash and rerun* pattern introduced by [Little et al. 2010].
Among the most important functionalities of Crowd Processes is their sophisticated parameter system that supports the declaration of types and the definition of default values. Supported are all types of the Java/Scala language, including *other crowd processes*, which allows for the creation of *nested processes*. Many processes, e.g., require a DECIDE-type process to select the best option among multiple candidates.
Every process exposes its mandatory and optional parameters as well as their defaults and validity bounds. This list of parameters is later used by the Recombinator to generate recombinations, which then essentially finds all possible definitions for these declarations. In case of nested processes, recombination therefore amounts to a recursive traversal of such a dependency tree.

4.3 The PPLib Process Repository Content

We populated the PPLib Process Repository with crowd processes gathered from the crowdsourcing literature. It currently contains 15 crowd processes out of which 5 do not nest any other process, and are therefore referred to as *basic building blocks*.

For the CREATE building blocks, we created two processes: (i) a *simple collection*, where the developer specifies the number of crowd workers that are asked to perform a certain task, and their outputs is returned in a list and (ii) a *collection with six-σ pruning*, as proposed by [Minder and Bernstein 2012b].

DECIDE has four basic building blocks: (i) *majority voting* (ii) *majority voting with N winners*, where elements that have been selected at least N times win, (iii) *Beat-By-K voting*, as introduced by [Goschin 2014], and (iv) *Statistical Reduction voting*, as implemented in [Barowy et al. 2012].

The nested processes for CREATE currently implemented in PPR are:

- *Iterative Refinement*: Iteratively asks crowd workers questions (using any CREATE element) and decides (using any DECIDE element) whether to use their answers or keep the previous state for the next iteration. The maximal number of iterations can be configured as a parameter.
- *Dual Pathway*: An adaption of [Liem and Chen 2011] with a DECIDE step.
- *CollectDecide*: Any CREATE process followed by any DECIDE process.
- *FindAndFix*: A DECIDE process with multiple winners followed by a CREATE Process.

The PPR can be extended by every user directly due to its Open Source nature. Extensions can be made available to the public by either creating an open GitHub repository specifically for the process[13], that is then easily importable by anyone as

[13] Note that this is <u>not the same</u> as creating a fork of the PPLib repository. Instead, one only publishes the code for the new process on his GitHub repository within an SBT module. This allows PPLib developers to essentially mix in as many 3rd party processes as they need while still using our official PPLib version. Note that other portals and modules could be added and mixed in the same way.

an SBT[7] dependency, or by sending the code to the official PPLib repository via a GitHub Pull Request.

4.4 Example: Using the PPLib Recombination Engine for Text Shortening

In this section we illustrate the use of the PPLib methodology by employing the PPLib framework for an actual use case. To that end, we will briefly walk through the four steps of the methodology and explain how they can be implemented in practice.

As an example use case, we are interested in shortening a book with 1000 pages. Our first step is to take a small sample of the book and use that sample to find the best crowd process to apply to the remainder of book.

STEP 1 – Intelligence: Identifying the process deep structure. After taking a sample of the book, we need to define the deep structure for the shortening process as well as a utility function. Algorithm 1 shows a possible implementation of the deep structure in Scala-code. It captures the core actions of text shortening: (i) dividing the sample into its paragraphs, (ii) obtaining a crowd process to shorten the paragraphs, (iii) running this process and then (iv) returning the result.

ALGORITHM 1. Process deep structure for text shortening

```
case class ShortNResult(text:String, costInCents:Int, durationInSeconds:Int) extends
   Comparable[ShortNResult] {

   override def compareTo(o:ShortNResult) = [..] // put utility function here
}

// The deep structure needs to implement PPLib’s DeepStructure interface. SimpleDeepStructure
// can be used for deep structures that only require 1 recombined crowd process.
// The DeepStructure interface requires a definition of the input type applied to the deep
// structure as well as the output type. In this case, the deep structure uses a string
// input (text) and returns a ShortNResult as output (see above, captures resulting text, cost
// and execution duration)
class ShortNDeepStructure extends SimpleDeepStructure[String, ShortNResult] {

   override def run(input:String, b:RecombinedProcessBlueprints) : ShortNResult = {
     // (i) Split the text (“data”) that was passed to this method into its paragraphs
     // (i.e., a list of patches, as required by inputType in obtainInstanceOfCrowdProcess)
     val paragraphs = IndexedPatch.from(input)

     // (ii) b will contain the blueprint for the recombined crowd process that we’ll apply
     // for text shortening. The first step is to create an instance of that process
     val shortenerProcess = obtainInstanceOfCrowdProcess(b)

     // (iii) supply the list of paragraphs to the process and let the crowd shorten it
     val shortenedParagraphs = shortenerProcess.process(paragraphs)

     // (iv) return result (shortened Text) with its cost and duration
     ShortNResult(shortenedParagraphs,
                  shortenerProcess.costSoFar,
                  shortenerProcess.durationSoFar)
   }

   def obtainInstanceOfCrowdProcess(b: RecombinedProcessBlueprints) = {
     //we’d like to pass the list of paragraphs to this process and get a shortened list
     //paragraphs in return. Therefore the desired input and output-types for this process
     //are the same and equal a list of patches (which is the type of our paragraphs-
     //member variable)
     type inputType = List[Patch]
     type outputType = inputType

     //create an instance of the process according to its deep structure and supply the
     //desired types. Return this instance
     b.createProcess[inputType, outputType]()
```

```
    }
    //[..] search space definition from ALGORITHM 2 here
}
```

The *run*-method receives the recombined processes as well as the text to be shortened as parameters and uses them to specify the process deep structure. (i) Since most processes in the PPR operate on a built-in data structure called *Patch*, the first step of the deep structure is to create patches from the supplied text sample. A *Patch* is an abstract entity that is used for any item that needs to be processed using PPLib. In this case, it represents a paragraph and its position in the text. (ii) The deep structure then creates an instance of the crowd process it will use and (iii) executes it, by calling the *process*-method and passing the prepared data structure with the list of patches resembling the paragraphs. (iv) Lastly, the result structure (ShortNResult) is constructed using the cost and duration of the process as well as the outcome of the process, i.e. the shortened paragraphs.

STEP 2 – Design: Defining parameter candidates. After defining the deep structure, we now need to instruct the Recombinator correctly, such that it can generate surface structures that apply to our use case of shortening paragraphs using crowds as shown in Algorithm 2. (i) In a first step, we define the applicable basic type from the PPR (CREATE / DECIDE) and the input and output types. (ii) We then define the crowd worker instructions that are to be used in the target process and (iii) bind these instructions to all generated processes. (iv) Our last step is to put everything together in a search space definition and return it.
Note that this all takes place in the same class as the one that we've used in step 1.

ALGORITHM 2. Defining the search space for the Recombinator

```
import ch.uzh.ifi.pdeboer.pplib.process.recombination.RecombinationHints._

class ShortNDeepStructure extends SimpleDeepStructure[String, ShortNResult] {
    // ... run method as shown in algorithm 1
    override def defineSimpleRecombinationSearchSpace = {
        // (i) Our target process should be a Create-type process that takes any descendant
        // of a list of patches as input (in Scala notation: "_ <: List[Patch]") and output.
        // This includes specializations of lists and/or the Patch-class.
        type targetProcessType = CreateProcess[_ <: List[Patch], _ <: List[Patch]]

        // (ii) The most important restriction on the search space is to define the crowd
        // task. By using "InstructionData" we also specify how a question gets rendered
        // to the crowd. On a process, an InstructionGenerator (IG) will then phrase
        // instructions in a way that fits the process. For example, in a majority vote
        // process, the IG will phrase a question out of this base data to pick the best
        // shortened paragraph while paying attention to grammar and text-length.
        val instructions = RecombinationHints.instructions(List(
            new InstructionData(actionName = "shorten the following paragraph",
                detailedDescription = "grammar (e.g., tenses), text-length")
        ))

        // (iii) Specify that we'd like to use the default hints for all building blocks that
        // the Recombinator processes. It would be possible to target specific building
        // blocks and supply different hints to them.
        // The default hints are just the instructions specified above.
        val hints = RecombinationHints.create(Map(DEFAULT_HINTS -> instructions))

        // (iv) Construct the search space and return it
        RecombinationSearchSpaceDefinition[targetProcessType](hints)
    }
}
```

(i) We first define the target type from the PPR that we'd like to use (CREATE). This Create Process should be able to work on a list of patches as input as well as a list of patches as output.

(ii) We then define the instructions that are used to generate questions to crowd workers. PPLib can generate instructions automatically based on provided core formulations and adapt them to the kind of process that uses the instructions (CREATE / DECIDE). For our initial experiments, we have used a simple instruction generator, that inserts different predefined text for CREATE and DECIDE to make a question out of the core instruction data. It is possible to extend PPLib with other, more complex instruction generators. A more detailed explanation on instruction generation is provided on our GitHub repository[14].
(iii) After specifying the formula to generate instructions for crowd workers, we use it as the only default parameter and (iv) construct the search space using instructions and the target type definition.

STEP 3 – Choice: Recombination and Auto-Experimentation. This step brings it all together: In the executable Main class of our PPLib project (see Algorithm 3), (i) we first create an instance of the deep structure as specified in step 1. (ii) The deep structure is then passed to the Recombinator who uses it to generate surface structures of our process, i.e., its concrete implementations. (iii) Lastly, the surface structures are supplied to the auto-experimentation engine and executed there. The auto-experimentation engine uses the utility function as defined in step 1 to determine the best performing process among all the surface structures.

ALGORITHM 3. Recombination and Auto-Experimentation

```
object ShortnText extends App {
    val text = "shorten me [..]" //text that needs to be shortened. E.g., could be loaded from PDF

    val deepStructure = new ShortNDeepStructure //(i) instantiate step 1 & 2 from above

    //(ii) use the search space defined in the deep structure to generate the surface structures
    val surfaceStructures = new Recombinator(deepStructure).recombine()

    //(iii) create AutoExperimentationEngine and let it run one iteration on all surface structures.
    //Note, that one can easily filter the surface structures before running auto-experimentation.
    val results = new AutoExperimentationEngine(surfaceStructures).runOneIteration(text)

    //use utility function to find the process with the best result. Alternatively, one can also
    //export the result and pick the best one using one's optimization tool of choice.
    val bestProcess = results.bestProcess

    println(s"The best process is $bestProcess")
}
```

5. EVALUATION

In this section we evaluate our claims by showing the usefulness of the PPLib methodology and library through the implementation of two non-trivial but typical crowd-processing examples – text translation and text shortening.
By the time of submission, PPLib was used to process more than 19'400 micro tasks involving more than 900 unique workers (by Turker ID). All the micro tasks submitted for this evaluation were restricted to experienced[15] workers from the United States. Crowd worker compensation depended on the time allotted for a crowd task[16].

[14] https://github.com/uzh/PPLib/blob/master/docs/instructiongenerator.md
[15] Workers with less than 4% rejected hits, and more than 4000 approved hits
[16] We care about crowd workers and are happy to report to have a flawless 5/5 score on all dimensions on Turkopticon at the time of submission

5.1 Text translation

In order to evaluate the usefulness of the PPLib methodology and library, we have applied it on a German-English translation. We used an article of the Swiss bank UBS' news portal available in both, German and English, with the English version serving as our benchmark[17]. The article consists of 209 words in 16 sentences. The algorithm's deep structure was inspired by the pattern used in [Minder and Bernstein 2012a] and first translated the article from German to English using Google Translate as a baseline. The resulting text was then split into its paragraphs (2-3 sentences each), which were then refined in a CREATE-type task by crowd workers. Afterwards, the refinements were collected and aggregated into a full article, based on their original positions.

For our experiments we restricted parameter recombination to the testing of two worker counts (i.e., 5 and 7 for CREATE steps, 3 and 5 for DECIDE steps) and did not vary other parameters such as K for Beat-By-K. This resulted in the 8 basic building blocks as detailed in Table 1:

4 DECIDE variations	4 CREATE variations
○ MV3: Majority voting with 3 voters	○ SC5: A simple collection made by 5 workers
○ MV5: Majority voting with 5 voters	○ SC7: A simple collection made by 7 workers
○ BV: Beat-By-K Voting with a maximum of 20 votes cast	○ PC5: A collection with six-σ pruning made by 5 workers
○ SV: Voting with Statistical Reduction with a confidence level of 85% and a hard maximum of 20 votes cast	○ PC7: A collection with six-σ pruning made by 7 workers

Table 1: Building block variations used in the Translation experiment

The process recombination was unrestricted and lead to 6 nested processes, as shown in Figure 3: Nested Processes Used in Translation Experiment. Each nested process was recombined with all applicable building blocks from Table 1. Figure 3 indicates the number of applicable building blocks for each nested process in brackets (DECIDE is green, CREATE is grey, nesting is black)

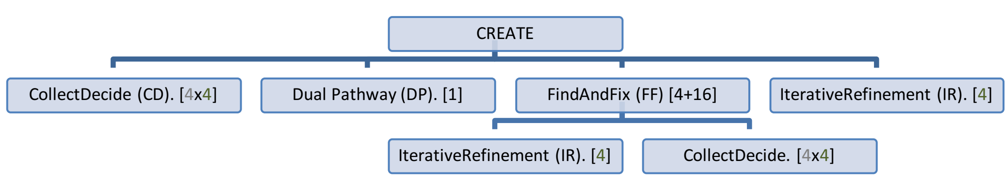

Figure 3: Nested Processes Used in Translation Experiment (number of applicable building blocks for each nested process in brackets, where DECIDE is green, CREATE is grey, nesting is black)

The search space definition, therefore, mostly exploited process recombination and lead the Recombinator to generate 41 surface structures (From left to right in Figure 3: CD (16) + DP (1) + FF(IR(4) + CD(16)) + IR(4) = 41).

[17] http://www.ubs.com/global/en/about_ubs/about_us/news/news.html/en/2014/12/29/consumption-indicator.html also available on our GitHub repository

After running all resulting 41 candidate processes once, we obtained 41 translations, which were then compared to the benchmark along 3 dimensions:

- *Readability*: How easy to understand are the resulting texts?
- *Adequacy*: The content of the reference translation is adequately represented by the translation algorithm.
- *Fluency*: The translated text should be grammatically correct and fluent.

Readability was automatically rated using the Flesch-Kincaid Grade Level [Kincaid et al. 1975] (normalized to values between 1-5), while adequacy and fluency were rated by 10 experts[18] recruited on eLance[19] on the 5 point Likert scale. Specifically, we took the 43[20] translations and randomly assigned them to the experts assuring that we would get 10 expert ratings per text (average rating μ shown in in Table 2). Additionally, we asked a professional translator from Switzerland to rate all resulting translations on both fluency and adequacy and compared his result with the average ratings of the expert crowd workers expecting a high correlation. The utility function was defined as the sum of all ratings. The top 5 processes in terms of the overall score are listed in Table 2.

Recombination	Adequacy (1 to 5)		Fluency (1 to 5)		Read-ability (1 to 5)	Utility value (3 to 15)
	μ	Professional	μ	Professional		
FF-PC5-BV	4.8	5	4.6	5	4.2	13.6
CD-SC5-SV	4.9	5	4.5	5	4.0	13.4
FF-SC7-MV3	4.8	5	4.3	5	4.2	13.3
FF-SC7-BV	4.7	5	4.5	5	4.0	13.2
CD-PC7-MV5	4.5	5	4.7	5	4.0	13.2
Benchmark	4.8	5	4.8	5	5.0	14.6
Baseline	4.3	5	3.9	5	3.8	12.0

Table 2: Ratings of the 5 top-performing recombinations for text-translation[21].

As anticipated, the Kendall-Tau correlation between the professional translator and the average rating of the expert crowds members was high (Adequacy: ~0.54; Fluency: ~0.59). We also found that the top 7 processes yielded results of equal quality compared to the benchmark in terms of accuracy according to a two-tailed T-Test with 95% significance level.

The recombination reaching the highest overall score of 13.6 consists of a *Find-step*, followed by a *Collection with six-σ pruning* whose best refinement is chosen in a *Beat-By-K voting process*. This particular composition of a crowd process has not been proposed before. In fact, most of the processes we identified as performing particularly well in the environment of text-translation have not previously been proposed and clearly exceed the performance of the original versions of well-known processes such as *Iterative Refinement* or *Dual Pathway*. This shows that the

[18] Excellent average grade on eLance (Higher than 4.5 points out of 5) as well as the eLance Qualification for the translation of German to English

[19] www.elance.com

[20] 41 translations from the candidate processes + gold standard + baseline (Google Translate) = 43

[21] Note that the ratings of the professional do not vary in the table, because the table shows only our 5 top-performing processes. The professional ratings indeed range from 3-5 over all processes

systematic exploration of the design space in text translation using Process Recombination provides significant improvements over the baseline of picking a well-known pattern among the ones proposed in the literature.

5.2 Text shortening

Shortening texts without loosing information is a central task in editing. For our second experiment we have built a crowd-task for text shortening and evaluated it akin to our previous evaluation. The code for the text shortening evaluation is similar to Algorithm 1-3 in section 4.4.

As example text, we used an English article from the popular news platform *the verge* with 1444 characters[22] and split it up into its individual paragraphs. Each paragraph was then processed in a Human Computation Task asking crowd workers to shorten the paragraph. We were eager to see whether the process that performed best in the translation task would be of comparable quality; we therefore reused the same selection of abstract processes for the recombination as we did in the translation example in section 5.1.

Additionally, all of the recombined candidates were executed four times on different days of the week in order to evaluate their stability as indicated by the standard deviation of their ranked performance across the four iterations. To avoid random results, we then continued with the 50% most stable recombinations and selected the 5 processes with the shortest median text length for a more in-depth evaluation. We then established their adequacy by selecting the shortest of their four results for each of these 5 processes, as we assumed it to be the one with the highest potential for content deviations from the baseline. Similarly to the translation-use case, these texts were then evaluated by 21[23] eLance crowd workers on the 5-point Likert scale.

Table 3 shows the 5 best performing, stable processes as evaluated by their median text length. The best process among these 5 shortened the article to 82% of its length in average.

Among the stable recombinations was a crowd process originally proposed in the Find-Fix-Verify paper [Bernstein et al. 2010]. The best-performing Find-Fix-Verify process shortened the text to an average of ~90% of its original length and was ranked 6th among our recombinations. In our experiments, however, we found the 5 recombinations shown in Table 3 that consistently outperformed Find-Fix-Verify in terms of text length as confirmed by a one-tailed Mann-Whitney test with $P < 0.001$.

Recombination	Text length (characters)		Adequacy (1 to 5)		Cost (US Dollars)	
	μ	σ	μ	σ	μ	σ
CD-SC5-MV5	1198.0	99.4	4.1	0.7	$4.25	$0.31
CD-SC5-MV3	1202.5	54.9	2.9	1.2	$3.45	$0.42
CD-SC5-BV	1278.5	82.9	3.3	1.1	$3.08	$0.38
CD-PC5-MV5	1281.0	68.3	4.1	0.9	$4.00	$0.29
IR-BV	1286.0	53.7	2.7	1.5	$5.28	$0.56

Table 3: Top 5 performing Recombinations for text shortening

[22] http://www.theverge.com/2015/1/8/7517361/google-getting-ready-to-sell-auto-insurance-and-maybe-buy-coverhound also available on our GitHub Repository

[23] 20 workers + 1 backup worker, in case someone doesn't provide his answer on time.

In summary, we showed that our approach also performs well when applied to text shortening: the systematic exploration and experimentation of the process design space yielded 5 stable recombinations that performed reliably better in terms of text length than the standard / baseline provided by Find-Fix-Verify.

5.3 Comparing the evaluations: Text Translation vs Text Shortening

Among our main claims for writing this paper is that a processes' performance varies in different applications and, therefore, one needs to explore the design space of all alternatives when changing the application domain. We verify this claim in this section by comparing the two evaluations above.

We first compared the score of processes in both our experiments. Specifically, we compared the ranking of stable text-shortening processes in terms of text length with their respective rank in the translation experiment as defined by their overall score. Kendall's Rank Correlation (τ_K) was τ_K =0.04 with P=0.82 indicating the complete absence of a relationship between the rankings of the two data sets.

Next, we investigated how stably specific building blocks performed in both experiments. To that end, we computed Kendall's Rank correlations for all recombined processes containing a specific building block. We found extremely unstable building blocks such as Majority Vote (τ_K=-0.05; P=0.88) or Collect-Decide (τ_K =-0.14; P=0.60), both giving strong evidence for an absence of a relationship between the performances in either setting.

We can, hence, confirm that good solutions–both in terms of overall process and in terms of process building blocks–are not stable across applications. These findings indicate that an approach based on best practices may not yield optimal results. In contrast, our methodology based on design-space exploration does not make such stability assumptions and can, therefore, find optimal processes in the absence of stability.

6. DISCUSSION

PPLib is well applicable to scenarios, where a task requestor has many predefined tasks that need to be completed by crowd workers in a process-like fashion. The PPLib approach imposes an upfront cost for finding the optimal process for the tasks at hand. On a high level, this cost is only worth bearing if the efficiency gain through using the optimal process makes up for the cost in the long run. For example, when processing the many predefined tasks the requestor would actually like to complete.

On a lower level, PPLib also introduces other advantages: Notably, it provides implementations for many well-known crowd patterns out of the box and doesn't require a potentially expensive programmer to code a crowd sourcing process all by herself. Therefore, we also see a timesaving component that should be taken into account for the cost-benefit analysis of PPLib.

The cost of using PPLib depends on the sample size of the tasks a requestor would like to complete and on the task complexity. A PPLib user specifies the money spent on every query sent out through PPLib and can, therefore, calculate the total expenses for processes with fixed query counts and specify a cost-ceiling for processes with dynamic query counts (e.g., Beat-By-K). We provided the functionality to do this automatically in PPLib by calling the method *getCostCeiling()* on the Recombinator after the search space definition.

7. LIMITATIONS & FUTURE WORK

It is easily understandable that generating all possible combinations of candidates leads to a search space that grows exponentially with every added parameter to be recombined. A complete exploration of the search space is therefore not possible with limited funds and limited (human/computational) processing power. Furthermore, in some domains, the complete generation of all recombinations may even be impossible, as it may lead to an infinite number of processes (e.g., due to continuous valued parameters). Hence, both the design space exploration via recombination and the auto-experimentation engine need to be improved: (i) The auto-experimentation engine should be extended to consider the robustness of results (i.e., the distribution of result quality), which is what we did manually in our evaluation. (ii) The current engine should be extended with findings from active sampling techniques, so as to avoid blindly sampling every process *i* times, and, instead, choose samples strategically. This is especially important in situations that face a certain experimentation budget [Saar-Tsechansky and Provost 2001; Sheng et al. 2008]. (iii) Integrating the auto-experimentation engine and the Recombinator will allow us to interleave (and trade-off) the exploration of the design space (i.e., recombination of new processes) with the exploitation of candidate recombinations (i.e., experimentation). We hope that the combination of these improvements may extend our approaches' performance when exploring solutions in high-dimensional domains with a limited budget. (iv) The power of the recombination engine is dependent on the abstract processes included in the Process Repository and the deep structure devised by the user. Akin to HTN-planning, which is limited by the plan grammar, our approach is limited by the diversity of the PPR to explore the possibilities for implementing the deep-structure. Hence, we built the PPR to be easily extensible.

Another important consideration is that the quality of the results of PPLib is dependent on the appropriateness of the user-defined utility function. Additional research is needed to explore the suitability of utility functions to different tasks.

As always in such settings, our experimental results are limited by a number of threats. First, we only ran a limited number of texts, which potentially limits the generalizability of our results to other texts. However, when combined with the findings of [Minder and Bernstein 2012a], who ran a larger variety of texts through a much smaller number of manually recombined processes, we believe that our newly proposed methodology with its automated recombination approach based on a repository stands and that our findings should be stable also for other texts. Second, in a similar vein, we could not run experiments with all possible environments and crowdsourcing tasks. But, we believe that our selection of two typical tasks provides sufficient evidence that our approach merits great potential for generalizing to other applications. Last but not least, we experienced some variability of experimental results, due to the variability of worker performance. In the text shortening exploration, for example, we found that some processes did not effectively control for worker variance and, therefore, yielded results of highly varying quality. Adhering to standard practice, we tried to address this issue using appropriate experimental procedures and statistical tests.

Finally, our experiments were limited to objective tasks omitting a subjective task such as image labeling. Future experiments will, therefore, have to establish PPLib's generalization to other classes of tasks.

8. CONCLUSION

Today's crowd process design is based on reusing various proposed patterns from the literature – essentially a best practices approach – or some special insight of a crowd process designer. Especially in a situation, where one needs to run a specific crowd process repeatedly, it is very important for this crowd process to be as efficient and effective as possible – something we believe can not be achieved with the currently established approach.

In this paper we introduced PPLib, both as a methodology and as a system implementing this methodology. PPLib leverages a repository of crowd process fragments as well as the notions of recombination and auto-experimentation to systematically explore the design space of possible processes by generating and trying alternatives for Human Computation tasks.

We evaluated our system threefold: (i) *Text translation*, where English-speaking crowd workers were asked to refine a machine translation of a German news article. Our system autonomously generated various recombinations for the human computation task, which were then supplied to the auto-experimentation engine and yielded translations coming close to the benchmark of a translation produced by professionals. Our recombination engine unearthed various processes that have not been proposed before, among them the process whose performance in the environment of text translation was best in terms of adequacy, fluency and readability.
(ii) *Text shortening*, where our recombination mechanism generated 41 process candidates. Among the recombined processes were four variations of the popular Find-Fix-Verify pattern proposed by [Bernstein et al. 2010]. While the results indicated that Find-Fix-Verify performs efficiently, its text-length consistently exceeded the shortened texts of five other generated processes in four iterations. (iii) In a comparison of both of our experiments, we could conclude that high-quality crowd processes as well as crowd process fragments do not generalize between settings, indicating the lack of silver bullets in the crowdsourcing domain.

Given these findings, we believe that our approach, based on process recombination and auto-experimentation, is a first candidate for systematic crowd process design – a relatively new area that needs further exploration.

ACKNOWLEDGMENTS

We would like to sincerely thank Patrick Minder for his support in the evaluation of the crowd translations. We are also thankful for the support received from Dan Lowenthal, Tobias Grubenmann, Michael Feldman and the anonymous reviewers of this paper, whose input was central for improving the paper. This work was supported in part by the Swiss National Science Foundation (SNSF- Project: 200021-143411/1).

REFERENCES

Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. 2011. The Jabberwocky Programming Environment for Structured Social Computing. *Architecture* (2011), 53. DOI:http://dx.doi.org/10.1145/2047196.2047203

Jürgen Angele, Michael Kifer, and Georg Lausen. 2009. Ontologies in F-Logic. In *Handbook on Ontologies*. 45–70. DOI:http://dx.doi.org/10.1007/978-3-540-92673-3_2

Daniel Barowy, Charlie Curtsinger, Emery Berger, and Andrew McGregor. 2012. AutoMan: A platform for integrating human-based and digital computation. *Proc. ACM Int. Conf. Object oriented Program. Syst. Lang. Appl. - OOPSLA '12* (2012), 639. DOI:http://dx.doi.org/10.1145/2384616.2384663

S. Battle, A. Bernstein, H. Boley, and B. Grosof. 2005. Semantic web services language (SWSL). *W3C Memb.* (2005), 1–61.

Abraham Bernstein. 2000. How can cooperative work tools support dynamic group process? Bridging the specificity frontier. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW '00)*. 279–288. DOI:http://dx.doi.org/10.1145/358916.358999

Abraham Bernstein, Mark Klein, and Thomas W. Malone. 2012. Programming the global brain. *Commun. ACM* 55 (2012), 41. DOI:http://dx.doi.org/10.1145/2160718.2160731

Abraham Bernstein, Mark Klein, and Thomas W. Malone. 1999. The process recombinator: a tool for generating new business process ideas. In *ICIS '99 Proceedings of the 20th international conference on Information*. 178–192.

Abraham Bernstein, Foster Provost, and Shawndra Hill. 2005. Towards Intelligent Assistance for a Data Mining Process: An Ontology-based Approach for Cost-sensitive Classification. *IEEE Trans. Knowl. Data Eng.* 17 (2005), 503–518. DOI:http://dx.doi.org/10.1109/TKDE.2005.67

Michael S. Bernstein et al. 2010. Soylent : A Word Processor with a Crowd Inside. (2010), 313–322.

Noam Chomsky. 1965. *Aspects of the Theory of Syntax*,

Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. (2004). http://martinfowler.com/articles/injection.html

Michael Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. CrowdDB: answering queries with crowdsourcing. *SIGMOD '11 Proc. 2011 ACM SIGMOD Int. Conf. Manag. data* (2011), 1–12. DOI:http://dx.doi.org/10.1145/1989323.1989331

Sergiu Goschin. 2014. *Stochastic dilemmas: foundations and applications*. Rutgers University-Graduate School-New Brunswick.

M. Hammer and J. Champy. 1993. Business process re-engineering. *London: Nicholas Brealey* (1993).

Michael Hammer, W. Gerry Howe, Vincent J. Kruskal, and Irving Wladawsky. 1977. A Very High Level Language for Data Processing Applications. *Commun. ACM* 20, 11 (1977), 832–840.

Oana Inel, Khalid Khamkham, Tatiana Cristea, and Anca Dumitrache. 2014. CrowdTruth : Machine-Human Computation Framework for Harnessing Disagreement in Gathering Annotated Data. In *International Semantic Web Conference (ISWC)*.

J. Peter Kincaid, Robert P. Fishburne Jr, Richard L. Rogers, and Brad S. Chissom. 1975. *Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel*,

Ross D. King et al. 2009. The automation of science. *Science* 324 (2009), 85–89. DOI:http://dx.doi.org/10.1126/science.1165620

Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E. Kraut. 2011. CrowdForge: Crowdsourcing Complex Work. In *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*. 43–52. DOI:http://dx.doi.org/10.1145/2047196.2047202

Anand Kulkarni, Matthew Can, and Björn Hartmann. 2012. Collaboratively crowdsourcing workflows with turkomatic. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work - CSCW '12*. 1003. DOI:http://dx.doi.org/10.1145/2145204.2145354

Jintae Lee, George M. Wyner, and Brian T. Pentland. 2008. Process grammar as a tool for business process design. *MIS Q.* 32 (2008), 757–778. DOI:http://dx.doi.org/Article

Beatrice Liem and Yiling Chen. 2011. An Iterative Dual Pathway Structure for Speech-to-Text Transcription. In 37–42.

Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. 2010. TurKit: Human Computation Algorithms on Mechanical Turk. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology - UIST '10*. 57. DOI:http://dx.doi.org/10.1145/1866029.1866040

Benjamin Livshits and Todd Mytkowicz. 2014. Saving Money While Polling with InterPoll Using Power Analysis. In *Conference on Human Computation and Crowdsourcing*. 159–170.

Thomas W. Malone et al. 1999. Tools for inventing organizations: Toward a handbook of organizational processes. *Manage. Sci.* 45, 3 (1999), 425–443.

Thomas W. Malone et al. 2003. Tools for inventing organizations: Toward a handbook of organizational processes. In Thomas W. Malone, Kevin Crowston, & George Herman, eds. *Organizing Business Knowledge: The MIT Process Handbook*. Cambridge, MA: MIT Press.

Thomas W. Malone and Kevin Crowston. 1994. The interdisciplinary study of coordination. *ACM Comput. Surv.* 26 (1994), 87–119. DOI:http://dx.doi.org/http://doi.acm.org/10.1145/174666.174668

Thomas W. Malone, Kevin Crowston, and George A. Herman. 2003. *Organizing Business Knowledge: the MIT Process Handbook*, Thomas W. Malone, Robert Laubacher, and Chrysanthos Dellarocas. 2010. The collective intelligence genome. *IEEE Eng. Manag. Rev.* 38 (2010), 38. DOI:http://dx.doi.org/10.1109/EMR.2010.5559142

Patrick Minder and Abraham Bernstein. 2012a. CrowdLang: A Programming Language for the Systematic Exploration of Human Computation Systems. In Karl Aberer, Andreas Flache, Wander Jager, Ling Liu, Jie Tang, & Christophe Guéret, eds. *Social Informatics*. Springer Berlin Heidelberg, 124–137. DOI:http://dx.doi.org/10.1007/978-3-642-35386-4_10

Patrick Minder and Abraham Bernstein. 2012b. CrowdLang: A Programming Language for the Systematic Exploration of Human Computation Systems. In Karl Aberer, Andreas Flache, Wander Jager, Ling Liu, Jie Tang, & Christophe Guéret, eds. *Social Informatics*. Springer Berlin Heidelberg, 124–137. DOI:http://dx.doi.org/10.1007/978-3-642-35386-4_10

D.S. Nau. 1987. Automated Process Planning Using Hierarchical Abstraction. *TI Tech. J.* (1987), 39–46.

Allen Newell and Herbert Alexander Simon. 1972. *Human problem solving*, Englewood Cliffs, N.J.,: Prentice-Hall.

Moo-ryong Ra, Bin Liu, Tom F. La Porta, and Ramesh Govindan. 2012. Medusa: a programming framework for crowd-sensing applications. *Proc. 10th Int. Conf. Mob. Syst. Appl. Serv. - MobiSys '12* , Section 2 (2012), 337. DOI:http://dx.doi.org/10.1145/2307636.2307668

Maytal Saar-Tsechansky and Foster Provost. 2001. Active learning for class probability estimation and ranking. *IJCAI Int. Jt. Conf. Artif. Intell.* (2001), 911–917.

John R. Searle. 1969. *Speech acts: an essay in the philosophy of language*, London,: Cambridge U.P.

Floarea Serban. 2010. Auto-experimentation of KDD workflows based on ontological planning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 313–320. DOI:http://dx.doi.org/10.1007/978-3-642-17749-1_22

Floarea Serban, Joaquin Vanschoren, Jörg-Uwe Kietz, and Abraham Bernstein. 2013. A Survey of Intelligent Assistants for Data Analysis. *ACM Comput. Surv.* 45 (2013), 31:1–31:35. DOI:http://dx.doi.org/10.1145/2480741.2480748

Victor S. Sheng, Foster Provost, and Panagiotis G. Ipeirotis. 2008. Get Another Label? Improving Data Quality and Data Mining Using Multiple, Noisy Labelers. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 614. DOI:http://dx.doi.org/10.1145/1401890.1401965

Herbert Alexander Simon. 1977. *The new science of management decision* Rev., Englewood Cliffs, N.J.: Prentice-Hall.

Stefano Tranquillini, Florian Daniel, and Pavel Kucherbaev. 2015. Modeling , Enacting , and Integrating Custom Crowdsourcing Processes. *ACM Trans. Web* 9, 2 (2015), 1–43.

Karl T. Ulrich and Steven D. Eppinger. 1995. *Product Design and Development*,

Vasilis Verroios and Michael S. Bernstein. 2014. Context Trees: Crowdsourcing Global Understanding from Local Views. In *HCOMP 2014.* Stanford InfoLab.

Terry Winograd and Fernando Flores. 1986. *Understanding Computers and Cognition: a New Foundation for Design*, Norwood, N.J.: Ablex Pub. Corp.

Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (1971), 221–227.

G.M. Wyner and Jintae Lee. 2002. Process specialization: defining specialization for state diagrams. *Comput. Math. Organ. Theory* 8 (2002), 133–155. DOI:http://dx.doi.org/10.1023/A:1016091900743

Michael D. Zisman. 1978. Office Automation: Revolution or Evolution? *Sloan Manage. Rev.* 19, 3 (1978), 1–16.